Performance Optimization of Multiprocessor Real-Time Applications by Graphs Transformations

C. Lavarenne and Y. Sorel *ParCo'93 – Grenoble, 7–9 September 1993*

INRIA, Domaine de Voluceau, Rocquencourt
B.P.105 - 78153 LE CHESNAY CEDEX
email: yves.sorel@inria.fr

Performance optimization of multiprocessor real-time applications is a resource allocation problem know to be NP-complete. A new heuristic optimizing the response time is presented. Graphs models are used to exhibit both the potential parallelism of the algorithm and the available parallelism of the multiprocessor. The heuristic is formalized in terms of successive graphs transformations. Its cost function is based on path calculi applied on execution durations of not only calculations but also inter-processor communications.

1. Introduction

Real-time signal processing applications require high performance computing. Signal processing is used here in its wide sense, including image processing and process control. Typical signal processing applications may be found in the following domains: biomedical (EEG and ECG processing...), telecoms (modems, echo cancelling, telephone network regulation...), automobile (engine control, ABS...), aerospace (aircraft and spacecraft control ...), process control (industrial and power plant regulation...), arms systems (radar, sonar, multi-sensor fusion...).

Each *application* is composed of a computer based *system* and of the *environment* it interacts with. The system itself is composed of a *hardware* part executing a *software* part. Finally the software part includes an application program, coding an *algorithm* independently from hardware considerations, and an *executive* allocating hardware resources in order to execute the application program. In these applications, the system is said to be *reactive* [1], because in order to keep control over its environment it must interact permanently with it, i.e. it must produce reactions to every stimulus coming from the environment. If not already discrete, input stimuli are sampled and called *input events* and reactions generate *output events*. The system is also said to be *real-time*, meaning that reactions must be produced within *bounded delays*.

Large application programs together with real-time constraints require high performance computing. Although new monoprocessors provide ever increasing computation power, they cannot cope with the ever increasing complexity of some signal processing applications. Parallel *multiprocessor* architectures are needed for such applications. In this

case, it must be pointed out that the executive has more resources to handle and therefore it induces more overhead which reduces the power benefit brought by multiprocessing. In this paper we consider only static executives, where the distribution and scheduling of the application program are decided before execution thus inducing low overhead. Then the overhead is mainly due to inter-processor communications which durations may not be neglected as far as real-time performances are concerned.

This paper is organized as follows. We first present the graphs models used to exhibit both the *potential parallelism* of the algorithm and the *available parallelism* of the multiprocessor. Then the implementation, i.e. distribution and scheduling of the algorithm on the multiprocessor, is formalized in terms of graphs transformations. Finally we state the performance optimization problem. Real-time constraints require an efficient implementation of the algorithm on the multiprocessor. Such an adequation is obtained by choosing among all the possible graph transformations one which optimizes the real-time performances. Real-time performances are mainly characterized by the input data rate and the response time, both expressed in our models in terms of critical path calculi. We give a response time optimization heuristic which is a "schedule flexibility" based greedy algorithm, upgraded to take into account inter-processor communication delays and critical path increase.

2. Algorithm and Multiprocessor Graph Models

In order to take advantage of the available parallelism of the multiprocessor, it is necessary to exhibit the potential parallelism of the algorithm. This may be obtained either directly from a data-flow program or extracted by data-dependence analysis of a sequential program.

The algorithm is therefore modelized by a data-flow graph (oriented hypergraph), that we call *software graph*, where each vertice is an *action* and each edge is a *data-flow*. Each action stands for either a calculation or a state memory or an external I/O whereas each edge stands for an iterative data transfert which induces a data dependence between the producing and consumming actions. On the other hand, the inputs of a calculation action must precede its outputs, this involves also a dependence. A memory action holds data in sequential order between iterations, there is no dependence between its input and its output. Therefore a memory action may be seen as two separate vertices, one without predecessor and the other without successor, then a graph with cycles only through memory vertices may be considered *acyclic*. External I/O actions without predecessor stand

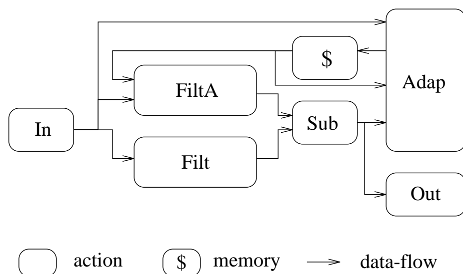


Figure 1. Software graph example

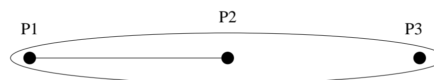


Figure 2. Hardware graph example

for the input interface handling the events produced by the environment. Symetrically the external I/O actions without successor stand for the output interface generating the reactions to the events. The whole dependencies of the acyclic graph induce a partial order on its actions. This model exhibits the potential parallelism of the algorithm.

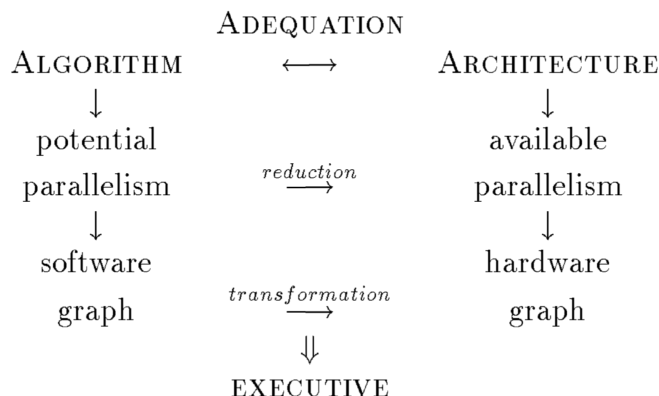
The multiprocessor, a MIMD or SPMD processor network, is modelized by a non oriented hypergraph, that we call *hardware graph*, where each vertice is a processor and each edge is a physical bidirectionnal communication *link* allowing data transferts between the memories of the processors connected to the link.

3. Graphs transformations

The graph models used for the algorithm and the architecture specification lead to a formalization of the implementation in terms of graphs transformations. The optimization aspects of the adequation between both graphs will be the subject of the next section. Whatever type of optimization, the implementation consists in reducing the potential parallelism of the software graph into the available parallelism of the hardware graph. This graph transformation consists in:

- distributing the actions on the processors, which leads to inter-processor communications which are themselves distributed on the inter-processor links
- scheduling the actions which have been assigned to a processor and scheduling the communications which have been assigned to an inter-processor link

From this graph transformation, a real-time distributed executive is automatically generated, allowing the execution of the algorithm on the multiprocessor. This issue has already been discussed in [2].



More formally, the distribution consists in partitionning the software graph in as many partition elements as there are processors. The inter-partition edges imply inter-processor communications. Though the hardware graph is inevitably connex, each processor is not necessarily directly connected to all the others. In order to allow communications between non directly connected processors, the hardware graph is first transformed into a

completely connected graph where the vertices are the same as in the hardware graph (the processors) and where the edges are all the paths of the hardware graph, called *routes*. Therefore any communication between any two processors will be assigned to a route.

The hardware graph is a pair $(\mathcal{P}, \mathcal{L})$ where $\mathcal{P}$ denotes the set of processors and $\mathcal{L}$ the set of inter-processor physical communication links. We then denote $\mathcal{R}$ the set of all the paths in $(\mathcal{P}, \mathcal{L})$. For instance, the graph $(\{p, p', p''\}, \{\{p, p'\}, \{p', p''\}\})$ has three routes, {({p, p'}), ({p', p"}), ({p, p'}, {p', p"})}. Note that routes, as links, are bidirectionnal. We call *routing* this transformation of the hardware graph into a routed hardware graph:

$$(\mathcal{P}, \mathcal{L}) \stackrel{routing}{\longrightarrow} (\mathcal{P}, \mathcal{R})$$

The software graph is a pair $(\mathcal{A}, \mathcal{D})$ where $\mathcal{A}$ denotes the set of actions and $\mathcal{D}$ the set of inter-action data-dependences. We then denote $\mathcal{A}_p$ the set of the actions assigned to $p \in \mathcal{P}$, by $\mathcal{D}_p$ the set of the data-dependences between actions belonging to $\mathcal{A}_p$ and by $\mathcal{D}_r$ the set of the inter-processor data-dependences generating communications assigned to $r \in \mathcal{R}$. We call $distrib_R$ the transformation of the software graph into the routed hardware graph:

$$((\mathcal{A}, \mathcal{D}), (\mathcal{P}, \mathcal{R})) \stackrel{distrib_R}{\longrightarrow} \left(\bigcup_{p \in \mathcal{P}} (\mathcal{A}_p, \mathcal{D}_p), \bigcup_{r \in \mathcal{R}} \mathcal{D}_r \right)$$

At this point, we have considered only the limitation on the number of processors. In order to also take into account the limitation on the number of inter-processor communication links, each $d_r \in \mathcal{D}_r$ is subsituted by a linear graph with as many vertices as there are links on the route:

$$\forall r \in \mathcal{R}, \quad \forall d_r \in \mathcal{D}_r, \quad d_r \longrightarrow (d_p, c_l, d_{p'}, \ldots c_{l''}, d_{p''})$$

Each such new vertice c_l, that we call *communication action*, corresponds to a data transfert between the memories of two directly connected processors. Actually all the processors connected to the physical link cooperate (in a way that depends on the hardware) to execute the transfert, so we can consider that the communication action is distributed over the processors sharing the link. Therefore the new edges d_p are intra-processor and belong to $\mathcal{D}_p$.

We denote $\mathcal{C}_l$ the set of all the c_l assigned to the same $l \in \mathcal{L}$. Now the $distrib_R$ transformation may be rewritten as the graph transformation *distrib*:

$$((\mathcal{A}, \mathcal{D}), (\mathcal{P}, \mathcal{L})) \stackrel{distrib}{\longrightarrow} \left(\bigcup_{p \in \mathcal{P}} (\mathcal{A}_p, \mathcal{D}_p), \bigcup_{l \in \mathcal{L}} \mathcal{C}_l \right)$$

The transformations *routing* and *distrib* do not modify the software graph partial order, that we denote $\prec$. On each processor, a scheduling of the actions is a total order $<_p \subset \prec$ on $\mathcal{A}_p$. Identically on each link, a scheduling of the communication actions is a total order $<_l \subset \prec$ on $\mathcal{C}_l$. By adding these schedulings to the *distrib* transformation, we finally obtain the transformation *dist/sched* which distributes and schedules the software graph on the hardware graph:

$$((\mathcal{A}, \mathcal{D}), (\mathcal{P}, \mathcal{L})) \stackrel{dist/sched}{\longrightarrow} \left(\bigcup_{p \in \mathcal{P}} (\mathcal{A}_p, \mathcal{D}_p, <_p), \bigcup_{l \in \mathcal{L}} (\mathcal{C}_l, <_l) \right)$$

4. Performance optimization

Several *dist/sched* transformations may be obtained from a software/hardware graphs pair. As stated in the introduction, we look for one which minimizes the response time of the transformed graph. The response time is the critical path of the graph, computed from the execution durations of the actions (including communication actions).

We denote $\Delta : \mathcal{A} \rightarrow \mathbb{R}$ the function giving the execution duration of every action. Note that usual optimization schemes do not take into account communication delays whereas in our model each routed communication is detailed enough so that the duration of each communication action may be determined in function of the amount of data it transfers [3]. Other hardware considerations such as memory conflicts or pipelines may also be considered to obtain more accurate communication durations, but we do not address these issues in this paper.

We then denote $\bar{\prec}$ the partial order between actions after the *dist/sched* transformation:

$$\bar{\prec} = \prec \cup \left(\bigcup_{p \in \mathcal{P}} <_p \right) \cup \left(\bigcup_{l \in \mathcal{L}} <_l \right)$$

We also denote $s(a)$ and $e(a) = s(a) + \Delta(a)$ the start and end execution dates of an action $a \in \mathcal{A}$. The partial order $\bar{\prec}$ is easily translated into relations between execution dates:

$$\forall a, a' \in \mathcal{A} \quad a \bar{\prec} a' \Longrightarrow e(a) \leq s(a')$$

Then by denoting $\Gamma(a) = \{a' \in \mathcal{A} \mid a \bar{\prec} a'\}$ the successors of an action $a \in \mathcal{A}$ and $\bar{\Gamma}(a) = \{a' \in \mathcal{A} \mid a' \bar{\prec} a\}$ its predecessors, we may define its *earliest start from start* $S(a)$ and its *earliest end from start* $E(a)$ and finally the response time R:

$$\begin{aligned} S(a) &= \begin{cases} 0 & \text{if } \bar{\Gamma}(a) = \emptyset \\ \max_{a' \in \bar{\Gamma}(a)} E(a') & \text{otherwise} \end{cases} \\ E(a) &= S(a) + \Delta(a) \\ R &= \max_{a \in \mathcal{A}} E(a) \end{aligned}$$

The optimization problem, minimizing R, as other resource allocation optimization problems, is known to be NP-complete. Among the heuristics used to cope with this problem, the ones based on *schedule flexibilities* are known to give almost as good results as simulated annealing but are much faster [4]. For every action, its schedule flexibility $F(a)$ is defined using R and its *latest end from end* $\bar{E}(a)$ and its *latest start from end* $\bar{S}(a)$:

$$\begin{aligned} \bar{E}(a) &= \begin{cases} 0 & \text{if } \Gamma(a) = \emptyset \\ \max_{a' \in \Gamma(a)} \bar{S}(a') & \text{otherwise} \end{cases} \\ \bar{S}(a) &= \bar{E}(a) + \Delta(a) \\ F(a) &= R - S(a) - \bar{S}(a) \end{aligned}$$

Schedule-flexibility based heuristics are greedy algorithms where an action is assigned to a processor at each step. This action is chosen among the ones, called *candidates*, which have all their predecessors already assigned. The schedule flexibility is used as a cost function to select the best action/processor pair.

We have developped a new schedule-flexibility based heuristic using our models to take into account communication delays (see the discussion on the Δ function at the beginning of this section). Moreover we have extended the cost function to take into account the critical path increase of candidate assignments. Indeed, assigning a candidate action to a processor may lead to delay the candidate's execution. When comparing the assignments of a candidate on different processors one may obtain different earliest start dates. The later an action is executed, the least schedule flexibility it has, until it becomes critical. From this point the schedule flexibility remains null (it is a piecewise continuous function) but the critical path begins to increase. We therefore introduce the *schedule penalty* $P(a)$ which is also a piecewise continuous function and is the counterpart of the schedule flexibility which is a sclack before any response time increase. The difference between the schedule penalty and the schedule flexibility is then a continuous function that we call *schedule pressure* $\sigma(a)$:

$$\forall a \in \mathcal{A} \quad \sigma(a) = P(a) - F(a)$$

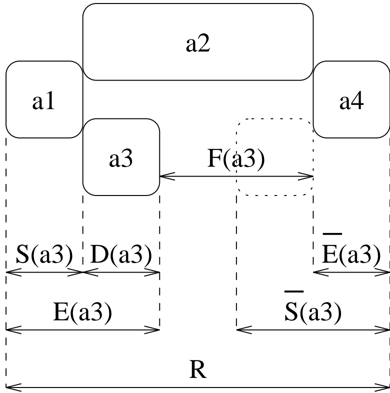


Figure 3. Path calculi

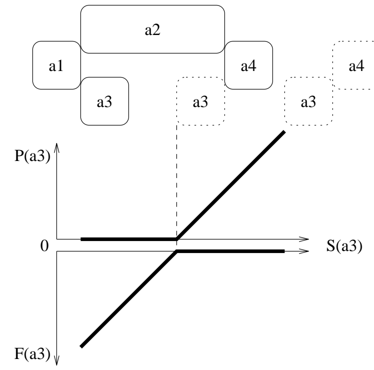


Figure 4. Schedule Flexibility and Penalty

With our models, the heuristic amounts to build step by step the *dist/sched* transformation by composing elementary graphs transformations. Each step transformation consists in assigning and scheduling a selected candidate action on a selected processor. Therefore, the number of steps is equal to the number of actions $n = \mathrm{Card}(\mathcal{A})$.

Before each step $i \in [1..n]$, we denote $\mathcal{A}_p^i \subseteq \mathcal{A}_p^{n+1} = \mathcal{A}_p$ the set of the actions already assigned on $p \in \mathcal{P}$, totally ordered by $\prec_p^i \subseteq \prec_p^{n+1} = \prec_p$. $\mathcal{C}_l^i$, $\prec_l^i$ and $\bar{\prec}^i$ are identically defined. Before the first step, we have initially empty sets except for $\bar{\prec}^1 = \prec$. The set of candidates is therefore:

$$V^i = \min_{\bar{\prec}^i} \left(\mathcal{A} - \bigcup_{p \in \mathcal{P}} \mathcal{A}_p^i \right)$$

At each step, every candidate/processor assignment is considered as possible next step, where the candidate is scheduled last on the processor on which it is assigned. Then we have the following relations where the iap indexed sets denote the eligible $(i+1)$ indexed sets:

$$\forall(a,p) \in V^i \times \mathcal{P} \quad \mathcal{A}_p^{iap} = \mathcal{A}_p^i \cup \{a\} \quad \text{and} \quad <_p^{iap} = <_p^i \cup \{ \left(\max_{<_p^i}(\mathcal{A}_p^i), a \right) \}$$

$$\forall(a,p) \in V^i \times \mathcal{P} \qquad \forall p' \in \mathcal{P} \mid p' \neq p \qquad \mathcal{A}_{p'}^{iap} = \mathcal{A}_{p'}^i \quad \text{and} \quad <_{p'}^{iap} = <_{p'}^i$$

For a given iap, for each resulting inter-processor communication with each of the candidate's predecessors, a communication action is created for each link of the shortest route from the predecessor's processor to the candidate's processor. For this purpose, we denote $\mathcal{R}_{pp'} \subseteq \mathcal{R}$ the set of all the routes between the processors p and p' and $\mathrm{Card}(r)$ the length (number of links) of any route $r \in \mathcal{R}$.

$$\bar{\gamma}^{iap} = \{a' \in \bar{\Gamma}(a) \mid a' \in \mathcal{A}_{p'}^i \quad p' \neq p\}$$

$$\forall a' \in \bar{\gamma}^{iap} \quad \exists p' \mid a' \in \mathcal{A}_{p'}^i \quad \exists r_{a'}^{iap} \in \mathcal{R}_{pp'} \quad \mid \quad \forall r \in \mathcal{R}_{pp'} \quad \mathrm{Card}\left(r_{a'}^{iap}\right) \leq \mathrm{Card}(r)$$

$$\forall l \in \mathcal{L} \qquad \mathcal{C}_l^{iap} = \mathcal{C}_l^i \bigcup_{a' \in \bar{\gamma}^{iap} \mid l \in r_{a'}^{iap}} \left\{ c_{a'l}^{iap} \right\}$$

The communication actions of a given inter-processor communication are totally ordered along the route from the sender action to the receiver action. Moreover the inter-processor communications are routed in the order of their earliest-start dates (i.e. the communication actions assigned on the same link are totally ordered by the earliest-end date of their sender action):

$$\forall \{a', a''\} \subset \bar{\gamma}^{iap} \quad \forall l \in \left(r_{a'}^{iap} \cap r_{a''}^{iap} \right) \qquad E(a') \leq E(a'') \Rightarrow c_{a'l}^{iap} <_l^{iap} c_{a''l}^{iap}$$

At this point, the schedule pressure σ^{iap} of every possible candidate/processor pair at step i may be determined. For a candidate, the best processor is the one which minimizes the schedule pressure, and among these (candidate, best processor) pairs, the best, elected at step i, is the one which maximizes the schedule pressure:

$$\exists(a,p) \in V^i \times \mathcal{P} \quad \mid \quad \sigma^{iap} = \max_{a' \in V^i} \left(\min_{p' \in \mathcal{P}} \sigma^{ia'p'} \right)$$

5. Conclusion

Thanks to our graph models and the associated graph transformations, we have been able to formalize precisely the response time used as performance criterion and to give a heuristic optimizing it. The heuristic gives a global solution obtained from local optimizations. As the result of local optimizations does not always lead to a global optimum (minimum response time), the user may specify *assignment constraints* to narrow the exploration space in order to direct a new optimization towards a better result. These constraints are then seen as initial conditions for the heuristic.

We are presently refining our hardware model in order to obtain a more accurate determination of calculation and inter-processor communications execution durations. The heuristic's results will benefit by these improvements. Future research axis are planned in the field of memory optimization, input rate minimization and its relationship with response time optimization.

We have developped a graphical development environment, called SynDEx [5] which uses the graphs models to specify both the algorithm and the multiprocessor, implements the heuritstic to optimize distribution and scheduling and finally generates automatically an optimized executive. With this environment, low level hand coding and multiprocessor real-time code debugging are eliminated, consequently the development cycle time of real-time applications is tremendously reduced. Moreover, as the optimization process is fast, several *hardware sizes* may be rapidly compared in order to find the best compromise between real-time and hardware size constraints.

REFERENCES

1. A. Benveniste, G. Berry:
   The Synchronous Approach to Reactive and Real-Time Systems.
   Proc. of the IEEE vol79, n.9, pp.1270–1282, 1991.
2. C. Lavarenne, Y. Sorel:
   Specification, Performance Optimization and Executive Generation for Real-Time Embedded Multiprocessor Applications with SynDEx.
   Proc. of Real-Time Embedded Processing for Space Applications, CNES International Symposium, 1992.
3. F. Ennesser, C. Lavarenne, Y. Sorel:
   Méthode chronométrique pour l'optimisation du temps de réponse des exécutifs SynDEx.
   INRIA Research Report n°1769, 1992.
4. C. Coroyer, Z. Liu:
   Effectiveness of heuristics and simulated annealing for the scheduling of concurrent tasks: an empirical comparison.
   INRIA Research Report n°1379, 1991.
5. C. Lavarenne, O. Seghrouchni, Y. Sorel, M. Sorine:
   The SynDEx software environment for real-time distributed systems design and implementation.
   Proc. of the European Control Conference, 1991.